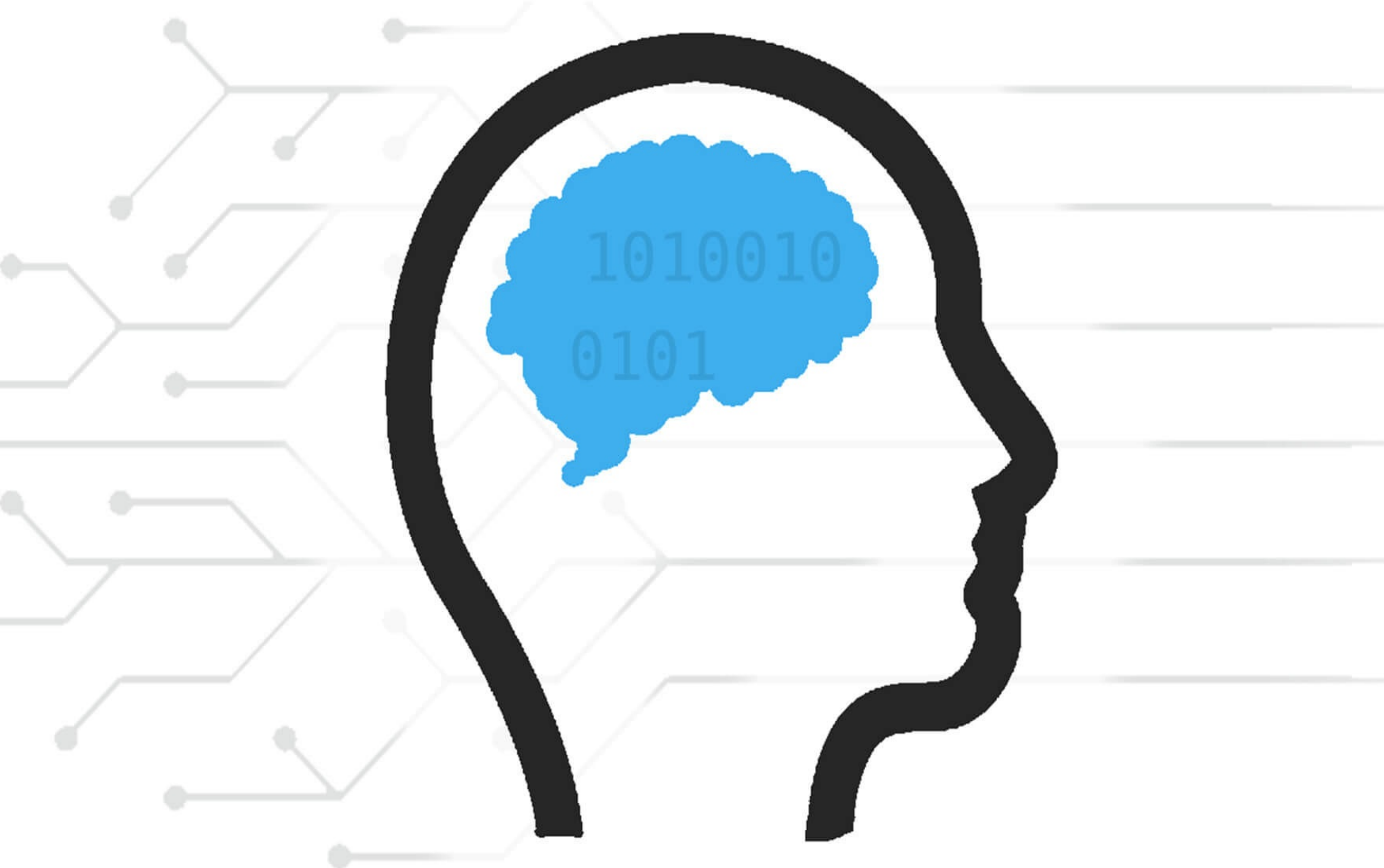


Algorithm

🄯 http://algorithm.yuanbin.me

Table of Contents

Data Structure and Algorithm/leetcode/lintcode

build passing slack 2/424 chat on slack

- English via Data Structure and Algorithm notes
- 简体中文请戳 数据结构与算法/leetcode/lintcode题解
- 繁體中文請瀏覽 資料結構與演算法/leetcode/lintcode題解

Introduction

This work is some notes of learning and practicing data structures and algorithm.

1. Part I is some brief introduction of basic data structures and algorithm, such as, linked lists, stack, queues, trees, sorting and etc.
2. Part II is the analysis and summary of programming problems, and most of the programming problems come from https://leetcode.com/, http://www.lintcode.com/, http://www.geeksforgeeks.org/, http://hihocoder.com/, https://www.topcoder.com/.
3. Part III is the appendix of resume and other supplements.

This project is hosted on https://github.com/billryan/algorithm-exercise and rendered by Gitbook. You can star the repository on the GitHub to keep track of updates. Another choice is to subscribe channel `#github_commit` via Slack https://ds-algo.slack.com/messages/github_commit/. ~~RSS feed is under development.~~

Feel free to access http://slackin4ds-algo.herokuapp.com for Slack invite automation.

You can view/search this document online or offline, feel free to read it. :)

- Online(Rendered by Gitbook): http://algorithm.yuanbin.me
- Offline(Compiled by Gitbook and Travis-CI):
  1. EPUB: GitHub, Gitbook, 七牛 CDN(中国大陆用户适用) - Recommended for iPhone/iPad/MAC
  2. PDF: GitHub, Gitbook, 七牛 CDN(中国大陆用户适用) - Recommended for Desktop
  3. MOBI: GitHub, Gitbook, 七牛 CDN(中国大陆用户适用) - Recommended for Kindle
- Site Search via Google: `keywords site:algorithm.yuanbin.me`
- Site Search via Swiftype: Click `Search this site` on the right bottom of webpages

License

Contribution

- English is maintained by @billryan
- 简体中文 is maintained by @billryan, @Shaunwei
- 繁體中文 is maintained by @CrossLuna

Other contributors can be found in Contributors to algorithm-exercise

Donation

本项目接受捐赠，形式不限，可以买书，可以寄明信片，也可以金额打赏 :)

邮寄明信片

@billryan 喜欢收集各种明信片，来者不拒~ 邮寄的话可以邮寄至 `上海市闵行区上海交通大学闵行校区电院群楼5号楼307` ，收件人： `袁斌` 。

送书

除了邮寄明信片，你还可以买本书送给各位贡献者，@billryan 的地址见上节。

支付宝

账户名：yuanbin2014(at)gmail.com 金额随意

Wechat

金额随意

PayPal

账户名：yuanbin2014(at)gmail.com 金额随意，付款时选择 friends and family

隐私考虑，以下名单隐去了部分个人信息，有些名单若没来得及添加，可私下联系我加上，有些信息和金额因为时间久远可能有误，欢迎指正。

- `taoli***@gmail.com` , 20
- `张亚*` , 6.66
- `wen***@126.com` , 20.16

- `she***@163.com` , 10
- `孙*` , 20
- `石*` , 50
- `文*` , 20
- `don***@163.com` , 5
- `129***@qq.com` , 50
- `130****9675` , 5
- `Tong W***` , 20 $
- `ee.***@gmail.com` , 6.66

所得捐款用于七牛 CDN 流量付费/激励 Contributors 写出更好的内容/购买书籍/西瓜/饮料

To Do

- [ ] add multiple languages support, currently 繁體中文, 简体中文 are available
- [x] explore nice writing style
- [x] add implementations of `Python` , `C++` , `Java` code
- [x] add time and space complexity analysis
- [x] summary of basic data structure and algorithm
- [x] add CSS for online website http://algorithm.yuanbin.me
- [x] add proper Chinese fonts for PDF output

FAQ - Frequently Asked Question

Some guidelines for contributing and other questions are listed here.

How to Contribute?

- Access Guidelines for Contributing for details.

Guidelines for Contributing

- Access English via Guidelines for Contributing
- 繁體中文請移步 貢獻指南
- 简体中文请移步 贡献指南

Part I - Basics

The first part summarizes some of the main aspects of data structures and algorithms, such as implementation and usage.

This chapter consists of the following sections.

Reference

- VisuAlgo - Animated visualizations of data structures and algorithms
- Data Structure Visualizations - An alternative to VisuAlgo
- Sorting Algorithms - Animations comparing various sorting algorithms

Data Structure

This chapter describes the fundamental data structures and their implementations.

String

String-related problems often appear in interview questions. In actual development, strings are also frequently used. Summarized here are common uses of strings in C++, Java, and Python.

Python

```python
s1 = str()
# in python, `''` and `""` are the same
s2 = "shaunwei" # 'shaunwei'
s2len = len(s2)
# last 3 chars
s2[-3:] # wei
s2[5:8] # wei
s3 = s2[:5] # shaun
s3 += 'wei' # return 'shaunwei'
# list in python is same as ArrayList in java
s2list = list(s3)
# string at index 4
s2[4] # 'n'
# find index at first
s2.index('w')  # return 5, if not found, throw ValueError
s2.find('w') # return 5, if not found, return -1
```

In Python, there's no StringBuffer or StringBuilder. However, string manipulations are fairly efficient already.

Java

```java
String s1 = new String();
String s2 = "billryan";
int s2Len = s2.length();
s2.substring(4, 8); // return "ryan"
StringBuilder s3 = new StringBuilder(s2.substring(4, 8));
s3.append("bill");
String s2New = s3.toString(); // return "ryanbill"
// convert String to char array
char[] s2Char = s2.toCharArray();
// char at index 4
char ch = s2.charAt(4); // return 'r'
// find index at first
int index = s2.indexOf('r'); // return 4. if not found, return -1
```

The difference between StringBuffer and StringBuilder is that the former guarantees thread safety. In a single-threaded environment, StringBuilder is more efficient.

Quick Sort

In essence, quick sort is an application of `divide and conquer` strategy. There are usually three steps:

1. Pick a pivot -- a random element.
2. Partition -- put the elements smaller than pivot to its left and greater ones to its right.
3. Recurse -- apply above steps until the whole sequence is sorted.

out-in-place implementation

Recursive implementation is easy to understand and code. Python `list comprehension` looks even nicer:

```
#!/usr/bin/env python


def qsort1(alist):
    print(alist)
    if len(alist) <= 1:
        return alist
    else:
        pivot = alist[0]
        return qsort1([x for x in alist[1:] if x < pivot]) + \
               [pivot] + \
               qsort1([x for x in alist[1:] if x >= pivot])

unsortedArray = [6, 5, 3, 1, 8, 7, 2, 4]
print(qsort1(unsortedArray))
```

The output：

```
[6, 5, 3, 1, 8, 7, 2, 4]
[5, 3, 1, 2, 4]
[3, 1, 2, 4]
[1, 2]
[]
[2]
[4]
[]
[8, 7]
[7]
[]
[1, 2, 3, 4, 5, 6, 7, 8]
```

Despite of its simplicity, above quick sort code is not that 'quick': recursive calls keep creating new arrays which results in high space complexity. So `list comprehension` is not proper for quick sort implementation.

Complexity

Take a quantized look at how much space it actually cost.

In the best case, the pivot happens to be the **median** value, and quick sort partition divides the sequence almost equally, so the recursions' depth is $\log n$. As to the space complexity of each level (depth), it is worth some discussion.

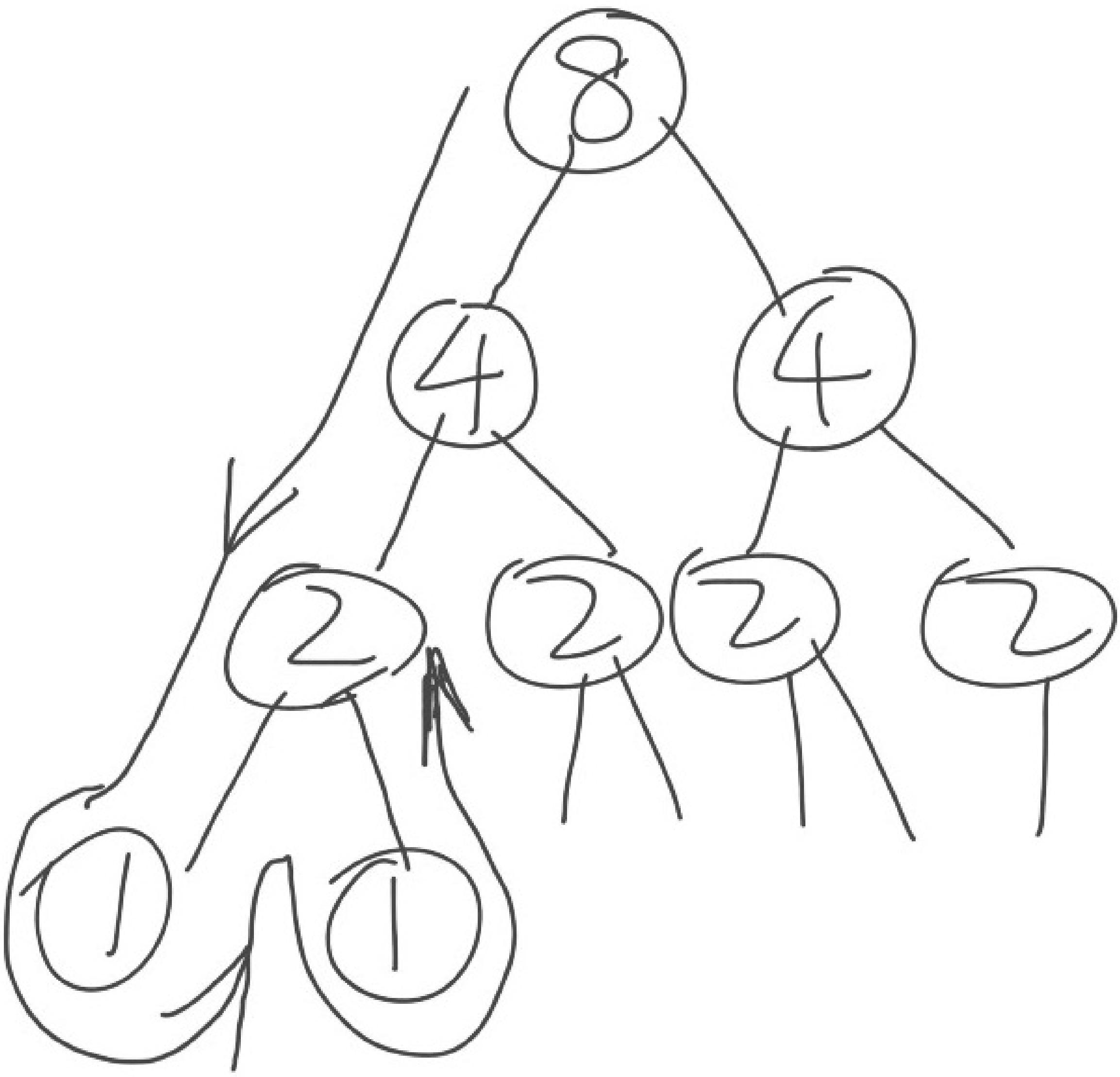
A common mistake can be: each level contains n elements, then the space complexity is surely $O(n)$. The answer is right, while the approach is not. As we know, space complexity is usually measured by memory consumption of a running program. Take above out-in-place implementation as example, **in the best case, each level costs half as much memory as its upper level does** . Sums up to be:

$\sum_{i=0} \frac{n}{2^i} = 2n$.

For more detail, refer to the picture below as well as above python code. The first level of recursion saves 8 values, the second 4, and so on so forth.

In the worst case, it will take $i-1$ times of swap on level i. Sums up to be:

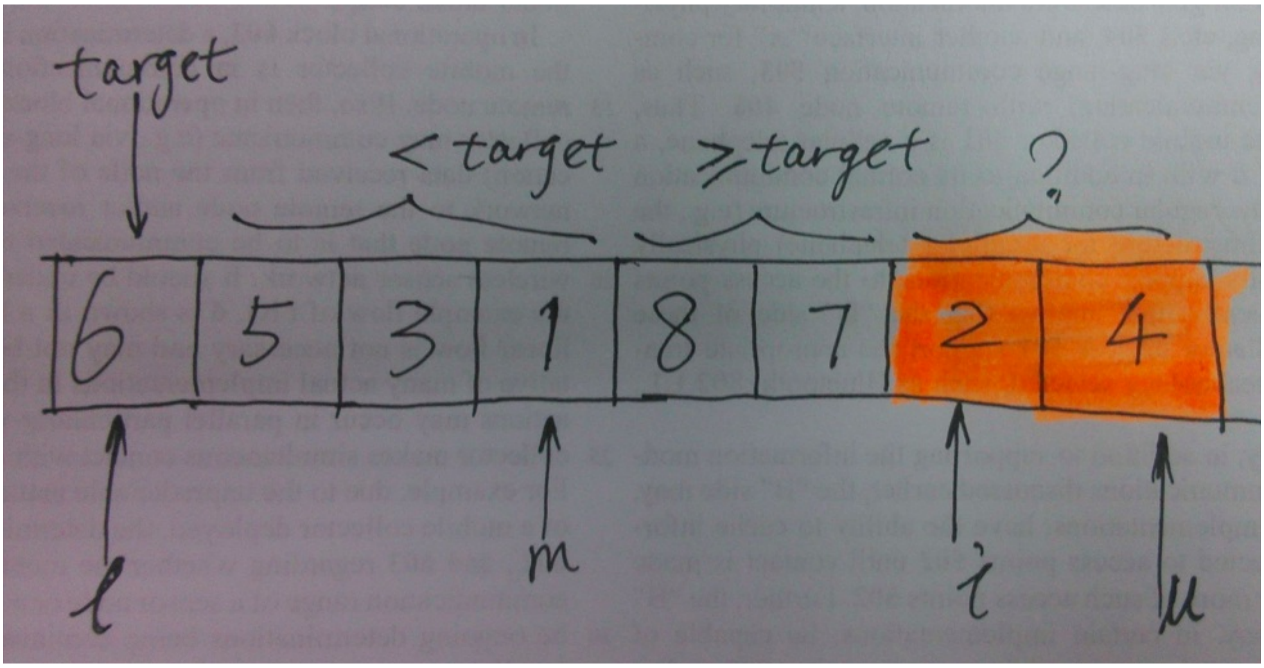
$\sum_{i=0}^{n}(n-i+1) = O(n^2)$



in-place implementation

one index for partition

One in-place implementation of quick sort is to use one index for partition, as the following image illustrates. Take example of `[6, 5, 3, 1, 8, 7, 2, 4]` again, l and u stand for the lower bound and upper bound of index respectively. i traverses and m maintains index of partition which varies with i. $target$ is the pivot.



For each specific value of i, $x[i]$ will take one of the follwing cases: if $x[i] \ge t$, i increases and goes on traversing; else if $x[i] < t$, $x[i]$ will be swapped to the left part, as statement `swap(x[++m], x[i])` does. Partition is done when `i == u` , and then we apply quick sort to the left and right parts, recursively. Under what circumstance does recursion terminate? Yes, `l >= u` .

Python

```
#!/usr/bin/env python


def qsort2(alist, l, u):
    print(alist)
    if l >= u:
        return

    m = l
    for i in xrange(l + 1, u + 1):
        if alist[i] < alist[l]:
            m += 1
            alist[m], alist[i] = alist[i], alist[m]
    # swap between m and l after partition, important!
    alist[m], alist[l] = alist[l], alist[m]
    qsort2(alist, l, m - 1)
    qsort2(alist, m + 1, u)

unsortedArray = [6, 5, 3, 1, 8, 7, 2, 4]
print(qsort2(unsortedArray, 0, len(unsortedArray) - 1))
```

Java

```
public class Sort {
    public static void main(String[] args) {
        int unsortedArray[] = new int[]{6, 5, 3, 1, 8, 7, 2, 4};
        quickSort(unsortedArray);
        System.out.println("After sort: ");
        for (int item : unsortedArray) {
            System.out.print(item + " ");
        }
    }

    public static void quickSort1(int[] array, int l, int u) {
        for (int item : array) {
            System.out.print(item + " ");
        }
        System.out.println();

        if (l >= u) return;
        int m = l;
        for (int i = l + 1; i <= u; i++) {
            if (array[i] < array[l]) {
                m += 1;
                int temp = array[m];
                array[m] = array[i];
                array[i] = temp;
            }
        }
        // swap between array[m] and array[l]
        // put pivot in the mid
        int temp = array[m];
        array[m] = array[l];
        array[l] = temp;

        quickSort1(array, l, m - 1);
        quickSort1(array, m + 1, u);
    }

    public static void quickSort(int[] array) {
        quickSort1(array, 0, array.length - 1);
    }
}
```

The swap of $x[i]$ and $x[m]$ should not be left out.

The output:

```
[6, 5, 3, 1, 8, 7, 2, 4]
[4, 5, 3, 1, 2, 6, 8, 7]
[2, 3, 1, 4, 5, 6, 8, 7]
[1, 2, 3, 4, 5, 6, 8, 7]
[1, 2, 3, 4, 5, 6, 8, 7]
[1, 2, 3, 4, 5, 6, 8, 7]
[1, 2, 3, 4, 5, 6, 8, 7]
[1, 2, 3, 4, 5, 6, 7, 8]
[1, 2, 3, 4, 5, 6, 7, 8]
```

Two-way partitioning

Another implementation is to use two indexes for partition. It speeds up the partition by working two-way simultaneously, both from lower bound toward right and from upper bound toward left, instead of traversing one-way through the sequence.

The gif below shows the complete process on `[6, 5, 3, 1, 8, 7, 2, 4]`.

6 5 3 1 8 7 2 4

1. Take `3` as the pivot.
2. Let pointer `lo` start with number `6` and pointer `hi` start with number `4`. Keep increasing `lo` until it comes to an element ≥ the pivot, and decreasing `hi` until it comes to an element < the pivot. Then swap these two elements.
3. Increase `lo` and decrease `hi` (both by 1), and repeat step 2 so that `lo` comes to `5` and `hi` comes to `1`. Swap again.
4. Increase `lo` and decrease `hi` (both by 1) until they meet (at `3`). The partition for pivot `3` ends. Apply the same operations on the left and right part of pivot `3`.

A more general interpretation:

1. Init i and j to be at the two ends of given array.
2. Take the first element as the pivot.
3. Perform partition, which is a loop with two inner-loops:
   - One that increases i, until it comes to an element ≥ pivot.
   - The other that decreases j, until it comes to an element < pivot.
4. Check whether i and j meet or overlap. If so, swap the elements.

Think of a sequence whose elements are *all equal*. In such case, each partition will return the middle element, thus recursion will happen $\log n$ times. For each level of recursion, it takes n times of comparison. The total comparison is $n \log n$ then. [programming_pearls]

Python

```
#!/usr/bin/env python

def qsort3(alist, lower, upper):
    print(alist)
    if lower >= upper:
        return

    pivot = alist[lower]
    left, right = lower + 1, upper
    while left <= right:
        while left <= right and alist[left] < pivot:
            left += 1
        while left <= right and alist[right] >= pivot:
            right -= 1
        if left > right:
            break
        # swap while left <= right
        alist[left], alist[right] = alist[right], alist[left]
    # swap the smaller with pivot
    alist[lower], alist[right] = alist[right], alist[lower]

    qsort3(alist, lower, right - 1)
    qsort3(alist, right + 1, upper)

unsortedArray = [6, 5, 3, 1, 8, 7, 2, 4]
print(qsort3(unsortedArray, 0, len(unsortedArray) - 1))
```

Java

```
public class Sort {
    public static void main(String[] args) {
        int unsortedArray[] = new int[]{6, 5, 3, 1, 8, 7, 2, 4};
        quickSort(unsortedArray);
        System.out.println("After sort: ");
        for (int item : unsortedArray) {
            System.out.print(item + " ");
        }
    }

    public static void quickSort2(int[] array, int l, int u) {
        for (int item : array) {
            System.out.print(item + " ");
        }
        System.out.println();

        if (l >= u) return;
        int pivot = array[l];
        int left = l + 1;
        int right = u;
        while (left <= right) {
            while (left <= right && array[left] < pivot) {
                left++;
            }
            while (left <= right && array[right] >= pivot) {
                right--;
            }
            if (left > right) break;
            // swap array[left] with array[right] while left <= right
            int temp = array[left];
            array[left] = array[right];
            array[right] = temp;
        }
        /* swap the smaller with pivot */
        int temp = array[right];
        array[right] = array[l];
        array[l] = temp;

        quickSort2(array, l, right - 1);
        quickSort2(array, right + 1, u);
    }

    public static void quickSort(int[] array) {
        quickSort2(array, 0, array.length - 1);
    }
}
```

The output:

```
[6, 5, 3, 1, 8, 7, 2, 4]
[2, 5, 3, 1, 4, 6, 7, 8]
[1, 2, 3, 5, 4, 6, 7, 8]
[1, 2, 3, 5, 4, 6, 7, 8]
[1, 2, 3, 5, 4, 6, 7, 8]
[1, 2, 3, 5, 4, 6, 7, 8]
[1, 2, 3, 4, 5, 6, 7, 8]
[1, 2, 3, 4, 5, 6, 7, 8]
[1, 2, 3, 4, 5, 6, 7, 8]
[1, 2, 3, 4, 5, 6, 7, 8]
[1, 2, 3, 4, 5, 6, 7, 8]
```

Having analyzed three implementations of quick sort, we may grasp one key difference between *quick sort* and *merge sort* :

1. Merge sort divides the original array into two sub-arrays, and merges the sorted sub-arrays to form a totally ordered one. In this case, recursion happens before processing(merging) the whole array.

2. Quick sort divides the original array into two sub-arrays, and then sort them. The whole array is ordered as soon as the sub-arrays get sorted. In this case, recursion happens after processing(partition) the whole array.

Robert Sedgewick's presentation on quick sort is strongly recommended.

Reference

- Quicksort - wikepedia
- Quicksort | Robert Sedgewick
- Programming Pearls Column 11 Sorting - gives an in-depth discussion on insertion sort and quick sort
- Quicksort Analysis
- > programming_pearls. Programming Pearls ↩

String

String related topics are discussed in this chapter.

> In order to re-use most of the memory of an existing data structure, internal implementation of string is immutable in most programming languages(Java, Python). Take care if you want to modify character in place.

strStr

Question

- leetcode: Implement strStr() | LeetCode OJ
- lintcode: lintcode - (13) strstr

Problem Statement

For a given source string and a target string, you should output the **first** index(from 0) of target string in source string.

If target does not exist in source, just return `-1`.

Example

If source = `"source"` and target = `"target"`, return `-1`.

If source = `"abcdabcdefg"` and target = `"bcd"`, return `1`.

Challenge

O(n2) is acceptable. Can you implement an O(n) algorithm? (hint: *KMP*)

Clarification

Do I need to implement KMP Algorithm in a real interview?

- Not necessary. When you meet this problem in a real interview, the interviewer may just want to test your basic implementation ability. But make sure your confirm with the interviewer first.

Problem Analysis

It's very straightforward to solve string match problem with nested for loops. Since we must iterate the target string, we can optimize the iteration of source string. It's unnecessary to iterate the source string if the length of remaining part does not exceed the length of target string. We can only iterate the valid part of source string. Apart from this naive algorithm, you can use a more effective algorithm such as KMP.

Python

```python
class Solution:
    def strStr(self, source, target):
        if source is None or target is None:
            return -1

        for i in range(len(source) - len(target) + 1):
            for j in range(len(target)):
                if source[i + j] != target[j]:
                    break
            else:  # no break
                return i
        return -1
```

C

```c
int strStr(char* haystack, char* needle) {
    if (haystack == NULL || needle == NULL) return -1;

    const int len_h = strlen(haystack);
    const int len_n = strlen(needle);
    for (int i = 0; i < len_h - len_n + 1; i++) {
        int j = 0;
        for (; j < len_n; j++) {
            if (haystack[i+j] != needle[j]) {
                break;
            }
        }
        if (j == len_n) return i;
    }

    return -1;
}
```

C++

```cpp
class Solution {
public:
    int strStr(string haystack, string needle) {
        if (haystack.empty() && needle.empty()) return 0;
        if (haystack.empty()) return -1;
        if (needle.empty()) return 0;
        // in case of overflow for negative
        if (haystack.size() < needle.size()) return -1;

        for (int i = 0; i < haystack.size() - needle.size() + 1; i++) {
            string::size_type j = 0;
            for (; j < needle.size(); j++) {
                if (haystack[i + j] != needle[j]) break;
            }
            if (j == needle.size()) return i;
        }

        return -1;
    }
};
```

Java

```java
public class Solution {
    public int strStr(String haystack, String needle) {
        if (haystack == null && needle == null) return 0;
        if (haystack == null) return -1;
        if (needle == null) return 0;

        for (int i = 0; i < haystack.length() - needle.length() + 1; i++) {
            int j = 0;
            for (; j < needle.length(); j++) {
                if (haystack.charAt(i+j) != needle.charAt(j)) break;
            }
            if (j == needle.length()) return i;
        }

        return -1;
    }
}
```

Source Code Analysis

1. corner case: `haystack(source)` and `needle(target)` may be empty string.

2. code convention:
    - space is needed for `==`
    - use meaningful variable names
    - put a blank line before declaration `int i, j;`
3. declare j outside for loop if and only if you want to use it outside.

Some Pythonic notes: 4. More Control Flow Tools section 4.4 and if statement - Why does python use 'else' after for and while loops?

Complexity Analysis

nested for loop, $O((n-m)m)$ for worst case.

Partition Array by Odd and Even

Question

- lintcode: (373) Partition Array by Odd and Even
- Segregate Even and Odd numbers - GeeksforGeeks

```
Partition an integers array into odd number first and even number second.

Example
Given [1, 2, 3, 4], return [1, 3, 2, 4]

Challenge
Do it in-place.
```

Solution

Use **two pointers** to keep the odd before the even, and swap when necessary.

Java

```java
public class Solution {
    /**
     * @param nums: an array of integers
     * @return: nothing
     */
    public void partitionArray(int[] nums) {
        if (nums == null) return;

        int left = 0, right = nums.length - 1;
        while (left < right) {
            // odd number
            while (left < right && nums[left] % 2 != 0) {
                left++;
            }
            // even number
            while (left < right && nums[right] % 2 == 0) {
                right--;
            }
            // swap
            if (left < right) {
                int temp = nums[left];
                nums[left] = nums[right];
                nums[right] = temp;
            }
        }
    }
}
```

C++

```
void partitionArray(vector<int> &nums) {
      if (nums.empty()) return;

      int i=0, j=nums.size()-1;
      while (i<j) {
          while (i<j && nums[i]%2!=0) i++;
          while (i<j && nums[j]%2==0) j--;
          if (i != j) swap(nums[i], nums[j]);
      }
  }
```

Src Code Analysis

Be careful not to forget `left < right` in while loop condition.

Complexity

To traverse the array, time complexity is $O(n)$. And maintaining two pointers means $O(1)$ space complexity.

Kth Largest Element in an Array

Tags: Quick Sort, Divide and Conquer, Medium

Question

- leetcode: (215) Kth Largest Element in an Array
- lintcode: (5) Kth Largest Element

Problem Statement

Find the **k**th largest element in an unsorted array. Note that it is the kth largest element in the sorted order, not the kth distinct element.

For example,
Given `[3,2,1,5,6,4]` and k = 2, return 5.

Note:
You may assume k is always valid, 1 ≤ k ≤ array's length.

Credits:

Special thanks to @mithmatt for adding this problem and creating all test cases.

Solution

Trail and error: Comparison-based sorting algorithms don't work because they incur ***O(n2)*** time complexity. Neither does Radix Sort which requires the elements to be in a certain range. In fact, Quick Sort is the answer to `kth largest` problems (Here are code templates of quick sort).

By quick sorting, we get the final index of a pivot. And by comparing that index with `K`, we decide which side (the greater or the smaller) of the pivot to recurse on.

Java - Recursion

```
public class Solution {
    public int findKthLargest(int[] nums, int k) {
        if (nums == null || nums.length == 0) {
            return Integer.MIN_VALUE;
        }

        int kthLargest = qSort(nums, 0, nums.length - 1, k);
        return kthLargest;
    }

    private int qSort(int[] nums, int left, int right, int k) {
        if (left >= right) {
            return nums[right];
        }

        int m = left;
        for (int i = left + 1; i <= right; i++) {
            if (nums[i] > nums[left]) {
                m++;
                swap(nums, m, i);
            }
        }
        swap(nums, m, left);

        if (k == m + 1) {
            return nums[m];
        } else if (k > m + 1) {
            return qSort(nums, m + 1, right, k);
        } else {
            return qSort(nums, left, m - 1, k);
        }
    }

    private void swap(int[] nums, int i, int j) {
        int tmp = nums[i]; nums[i] = nums[j]; nums[j] = tmp;
    }
}
```

Src Code Analysis

Two cases when the recursion ceases: a. left bound equals right bound; b. final index of pivot equals K.

Since 'Kth **largest**' is wanted, numbers greater than pivot are placed to the left and numbers smaller to the right, which is a little different with typical quick sort code.

Java - Iteration

Recursive code is easier to read than to write, and it demands some experience and skill. Here is an iterative implementation.

```
class Solution {
    public int findKthLargest(int[] A, int k) {
        if (A == null || A.length == 0 || k < 0 || k > A.length) {
            return -1;
        }

        int lo = 0, hi = A.length - 1;
        while (lo <= hi) {
            int idx = partition(A, lo, hi);
            if (idx == k - 1) {
                return A[idx];
            } else if (idx < k - 1) {
                lo = idx + 1;
            } else {
                hi = idx - 1;
            }
        }

        return -1;
    }

    private int partition(int[] A, int lo, int hi) {
        int pivot = A[lo], i = lo + 1, j = hi;
        while (i <= j) {
            while (i <= j && A[i] > pivot) {
                i++;
            }
            while (i <= j && A[j] <= pivot) {
                j--;
            }
            if (i < j) {
                swap(A, i, j);
            }
        }
        swap(A, lo, j);

        return j;
    }

    private void swap(int[] A, int i, int j) {
        int tmp = A[i];
        A[i] = A[j];
        A[j] = tmp;
    }
}
```

Src Code Analysis

The `while` loop in `findKthLargest` is very much like that in `binary search`. And `partition` method is just the same as quick sort partition.

Complexity

Time Complexity. Worse case (when the array is sorted): ***n + n - 1 + ... + 1 = O(n^2)*** . Amortized complexity: ***n + n/2 + n/4 + ... + 1 = O(2n)=O(n)*** .

Space complexity is ***O(1)*** .

Search in Rotated Sorted Array

Question

- leetcode: (33) Search in Rotated Sorted Array
- lintcode: (62) Search in Rotated Sorted Array

Problem Statement

Suppose a sorted array is rotated at some pivot unknown to you beforehand.

(i.e., `0 1 2 4 5 6 7` might become `4 5 6 7 0 1 2`).

You are given a target value to search. If found in the array return its index, otherwise return -1.

You may assume no duplicate exists in the array.

Example

For `[4, 5, 1, 2, 3]` and `target=1` , return `2` .

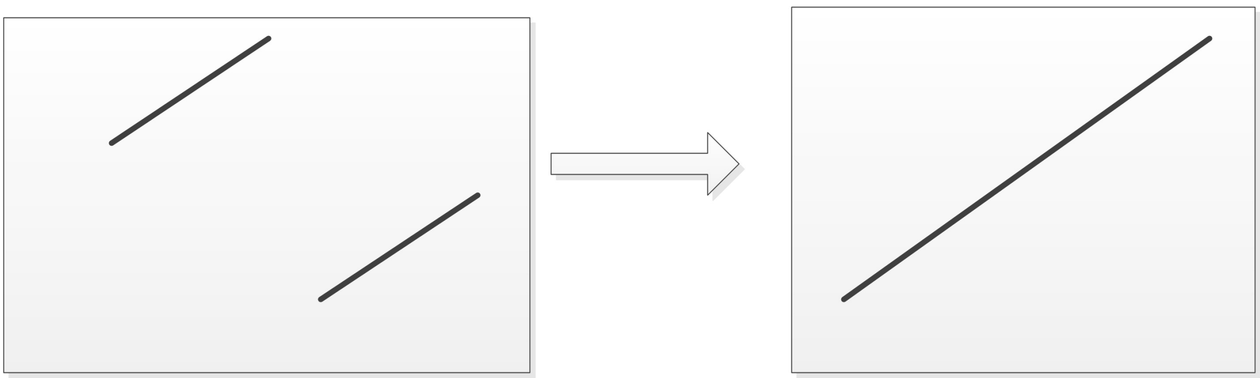
For `[4, 5, 1, 2, 3]` and `target=0` , return `-1` .

Challenge

O(logN) time

Solution1 - work on sorted subarray

Draw it. Rotated sorted array will take one of the following two forms:



Binary search does well in sorted array, while this problem gives an unordered one. Be patient. It is actually a combination of two sorted subarrayss. The solution takes full advantage of this. BTW, another approach can be comparing `target` with `A[mid]` , but dealing with lots of cases is kind of sophisticated.

C++

```
/**
 * 本代码fork自
 * http://www.jiuzhang.com/solutions/search-in-rotated-sorted-array/
 */
class Solution {
    /**
     * param A : an integer ratated sorted array
     * param target :  an integer to be searched
     * return : an integer
     */
public:
    int search(vector<int> &A, int target) {
        if (A.empty()) {
            return -1;
        }

        vector<int>::size_type start = 0;
        vector<int>::size_type end = A.size() - 1;
        vector<int>::size_type mid;

        while (start + 1 < end) {
            mid = start + (end - start) / 2;
            if (target == A[mid]) {
                return mid;
            }
            if (A[start] < A[mid]) {
                // situation 1, numbers between start and mid are sorted
                if (A[start] <= target && target < A[mid]) {
                    end = mid;
                } else {
                    start = mid;
                }
            } else {
                // situation 2, numbers between mid and end are sorted
                if (A[mid] < target && target <= A[end]) {
                    start = mid;
                } else {
                    end = mid;
                }
            }
        }

        if (A[start] == target) {
            return start;
        }
        if (A[end] == target) {
            return end;
        }
        return -1;
    }
};
```

Java

```java
public class Solution {
    /**
     *@param A : an integer rotated sorted array
     *@param target :  an integer to be searched
     *return : an integer
     */
    public int search(int[] A, int target) {
        if (A == null || A.length == 0) return -1;

        int lb = 0, ub = A.length - 1;
        while (lb + 1 < ub) {
            int mid = lb + (ub - lb) / 2;
            if (A[mid] == target) return mid;

            if (A[mid] > A[lb]) {
                // case1: numbers between lb and mid are sorted
                if (A[lb] <= target && target <= A[mid]) {
                    ub = mid;
                } else {
                    lb = mid;
                }
            } else {
                // case2: numbers between mid and ub are sorted
                if (A[mid] <= target && target <= A[ub]) {
                    lb = mid;
                } else {
                    ub = mid;
                }
            }
        }

        if (A[lb] == target) {
            return lb;
        } else if (A[ub] == target) {
            return ub;
        }
        return -1;
    }
}
```

Source Code Analysis

1. If `target == A[mid]` , just return.
2. Observe the two sorted subarrays, we can find that the least one of the left is greater than the biggest of the right. So if `A[start] < A[mid]` , then interval [start, mid] will be sorted.
3. Do binary search on `A[start] ~ A[mid]` on condition that `A[start] <= target <= A[mid]` .
4. Or do binary search on `A[mid]~A[end]` on condition that `A[mid] <= target <= A[end]` .
5. If while loop ends and none `A[mid]` hits, then examine `A[start]` and `A[end]` .
6. Return -1 if `target` is not found.

Complexity

The time complexity is approximately ***O(log n)***.

Solution2 - double binary search

Do binary search twice: first on the given array to find the break point; then on the proper piece of subarray to search for the target.

It may take a small step to see why the given array is binary-searchable. Though a rotated array itself is neither sorted nor monotone, there is implicit monotonicity. All elements on the left of break point are ≥A[0], and those on the right of break point are <A[0]. In a binary search, we keep narrowing the search scope by dropping the left or right half of the sequence,

and here in the rotated array, we can do that much similarly.

To formalize, define an array `A'` that `A'[i] = A[i] < A[0] ? true : false`. If `A` is `[4, 5, 6, 7, 0, 1, 2]`, `A'` will be `[false, false, false, false, true, true, true]`. Surely `A'` monotone.

Java

```
public class Solution {
    /**
     *@param A : an integer rotated sorted array
     *@param target :  an integer to be searched
     *return : an integer
     */
    public int search(int[] A, int target) {
        if (A == null || A.length == 0) {
            return -1;
        }

        int p = findBreakPoint(A);
        if (target >= A[0]) {
            // search in [lo, segPoint]
            return binSearch(A, target, 0, p);
        } else {
            // search in [segPoint, hi]
            return binSearch(A, target, p, A.length - 1);
        }
    }

    private int findBreakPoint(int[] A) {
        // A[index] < A[0], min[index]
        int index;

        int lo = 0, hi = A.length - 1, segValue = A[0];
        while (lo + 1 < hi) {
            int md = lo + (hi - lo)/2;
            if (A[md] > segValue) {
                lo = md;
            } else {
                hi = md;
            }
        }
        index = A[lo] < segValue ? lo : hi;

        return index;
    }

    private int binSearch(int[] A, int target, int lo, int hi) {
        while (lo + 1 < hi) {
            int md = lo + (hi - lo) / 2;
            if (A[md] == target) {
                lo = md;
            } else if (A[md] < target) {
                lo = md;
            } else {
                hi = md;
            }
        }

        if (A[lo] == target) {
            return lo;
        }
        if (A[hi] == target) {
            return hi;
        }
        return -1;
    }
}
```

Complexity

The first binary search costs ***O(log n)*** time complexity, and the second costs no more than ***O(log n)***.

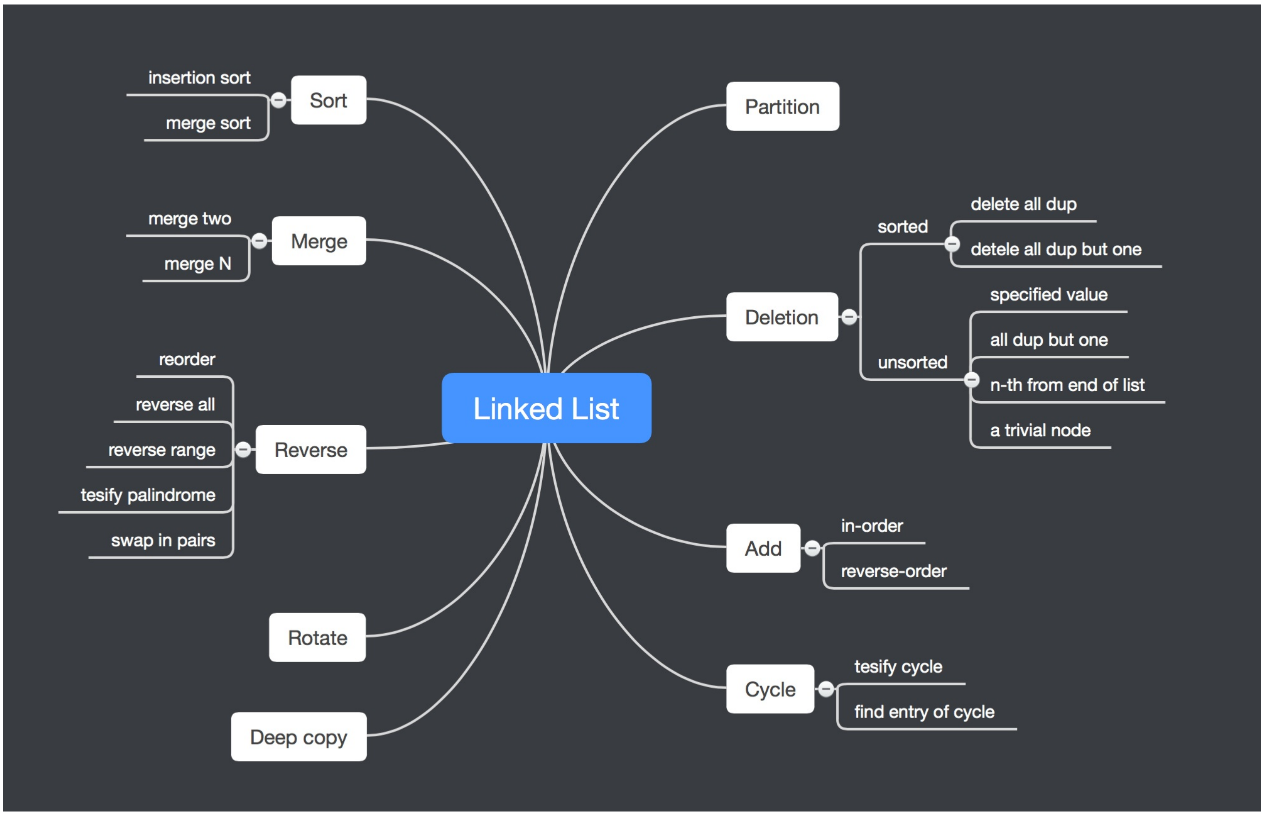
Linked List

This section includes common operations on linked list, such as deletion, insertion, and merging.

Frequently made mistakes:

- Not updating runner-node when traversing linked list
- Not recording head node before traversing
- returning incorrect pointer to node

The image below serves as a summarization of problems in this section.



Reverse Linked List

Question

- leetcode: (206) Reverse Linked List | LeetCode OJ
- lintcode: (35) Reverse Linked List

```
Reverse a linked list.

Example
For linked list 1->2->3, the reversed linked list is 3->2->1

Challenge
Reverse it in-place and in one-pass
```

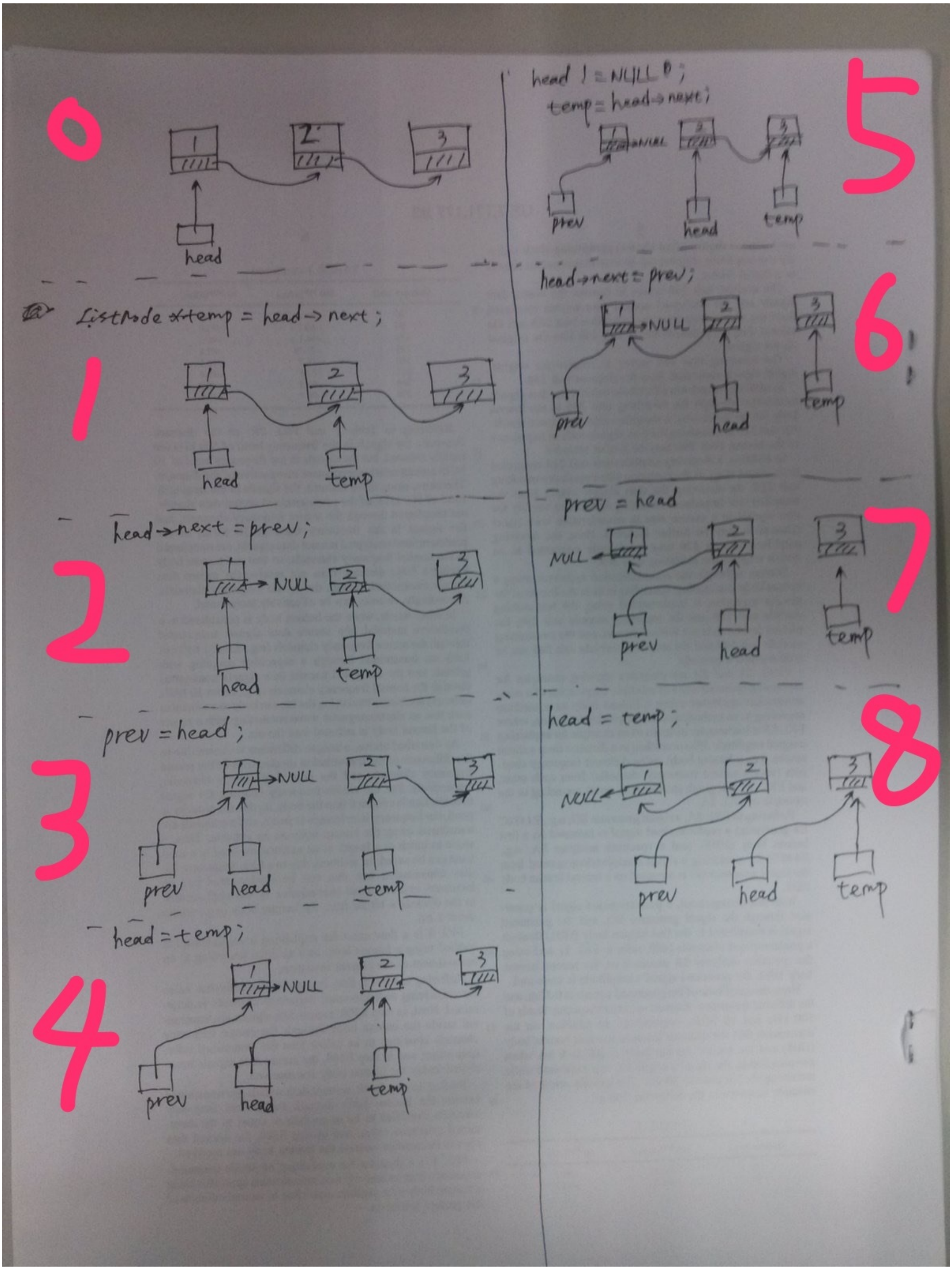
Solution1 - Non-recursively

It would be much easier to reverse an array than a linked list, since array supports random access with index, while singly linked list can ONLY be operated through its head node. So an approach without index is required.

Think about how '1->2->3' can become '3->2->1'. Starting from '1', we should turn '1->2' into '2->1', then '2->3' into '3->2', and so on. The key is how to swap two adjacent nodes.

```
temp = head -> next;
head->next = prev;
prev = head;
head = temp;
```

The above code maintains two pointer, `prev` and `head`, and keeps record of next node before swapping. More detailed analysis:



1. Keep record of next node
2. change `head->next` to `prev`
3. update `prev` with `head` , to keep moving forward
4. update `head` with the record in step 1, for the sake of next loop

Python

```python
# Definition for singly-linked list.
# class ListNode:
#     def __init__(self, x):
#         self.val = x
#         self.next = None

class Solution:
    # @param {ListNode} head
    # @return {ListNode}
    def reverseList(self, head):
        prev = None
        curr = head
        while curr is not None:
            temp = curr.next
            curr.next = prev
            prev = curr
            curr = temp
        # fix head
        head = prev

        return head
```

C++

```cpp
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode(int x) : val(x), next(NULL) {}
 * };
 */
class Solution {
public:
    ListNode* reverse(ListNode* head) {
        ListNode *prev = NULL;
        ListNode *curr = head;
        while (curr != NULL) {
            ListNode *temp = curr->next;
            curr->next = prev;
            prev = curr;
            curr = temp;
        }
        // fix head
        head = prev;

        return head;
    }
};
```

Java

```java
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
public class Solution {
    public ListNode reverseList(ListNode head) {
        ListNode prev = null;
        ListNode curr = head;
        while (curr != null) {
            ListNode temp = curr.next;
            curr.next = prev;
            prev = curr;
            curr = temp;
        }
        // fix head
        head = prev;

        return head;
    }
}
```

Source Code Analysis

Already covered in the solution part. One more word, the assignment of `prev` is neat and skilled.

Complexity

Traversing the linked list leads to ***O(n)*** time complexity, and auxiliary space complexity is ***O(1)***.

Solution2 - Recursively

Three cases when the recursion ceases:

1. If given linked list is null, just return.
2. If given linked list has only one node, return that node.
3. If given linked list has at least two nodes, pick out the head node and regard the following nodes as a sub-linked-list, swap them, then recurse that sub-linked-list.

Be careful when swapping the head node (refer as `nodeY`) and head of the sub-linked-list (refer as `nodeX`): First, swap `nodeY` and `nodeX` ; Second, assign `null` to `nodeY->next` (or it would fall into infinite loop, and tail of result list won't point to `null`).

Python

```python
"""
Definition of ListNode

class ListNode(object):

    def __init__(self, val, next=None):
        self.val = val
        self.next = next
"""
class Solution:
    """
    @param head: The first node of the linked list.
    @return: You should return the head of the reversed linked list.
                  Reverse it in-place.
    """
    def reverse(self, head):
        # case1: empty list
        if head is None:
            return head
        # case2: only one element list
        if head.next is None:
            return head
        # case3: reverse from the rest after head
        newHead = self.reverse(head.next)
        # reverse between head and head->next
        head.next.next = head
        # unlink list from the rest
        head.next = None

        return newHead
```

C++

```cpp
/**
 * Definition of ListNode
 *
 * class ListNode {
 * public:
 *     int val;
 *     ListNode *next;
 *
 *     ListNode(int val) {
 *         this->val = val;
 *         this->next = NULL;
 *     }
 * }
 */
class Solution {
public:
    /**
     * @param head: The first node of linked list.
     * @return: The new head of reversed linked list.
     */
    ListNode *reverse(ListNode *head) {
        // case1: empty list
        if (head == NULL) return head;
        // case2: only one element list
        if (head->next == NULL) return head;
        // case3: reverse from the rest after head
        ListNode *newHead = reverse(head->next);
        // reverse between head and head->next
        head->next->next = head;
        // unlink list from the rest
        head->next = NULL;

        return newHead;
    }
};
```

Java

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
public class Solution {
    public ListNode reverse(ListNode head) {
        // case1: empty list
        if (head == null) return head;
        // case2: only one element list
        if (head.next == null) return head;
        // case3: reverse from the rest after head
        ListNode newHead = reverse(head.next);
        // reverse between head and head->next
        head.next.next = head;
        // unlink list from the rest
        head.next = null;

        return newHead;
    }
}
```

Source Code Analysis

case1 and case2 can be combined.What case3 returns is head of reversed list, which means it is exact the same Node (tail of origin linked list) through the recursion.

Complexity

The depth of recursion: ***O(n)***. Time Complexity: ***O(N)***. Space Complexity (without considering the recursion stack): ***O(1)***.

Reference

- 全面分析再动手的习惯：链表的反转问题（递归和非递归方式） - 木棉和木槿 - 博客园
- data structures - Reversing a linked list in Java, recursively - Stack Overflow
- 反转单向链表的四种实现（递归与非递归，C++） | 宁心勉学，慎思笃行
- iteratively and recursively Java Solution - Leetcode Discuss

Tags